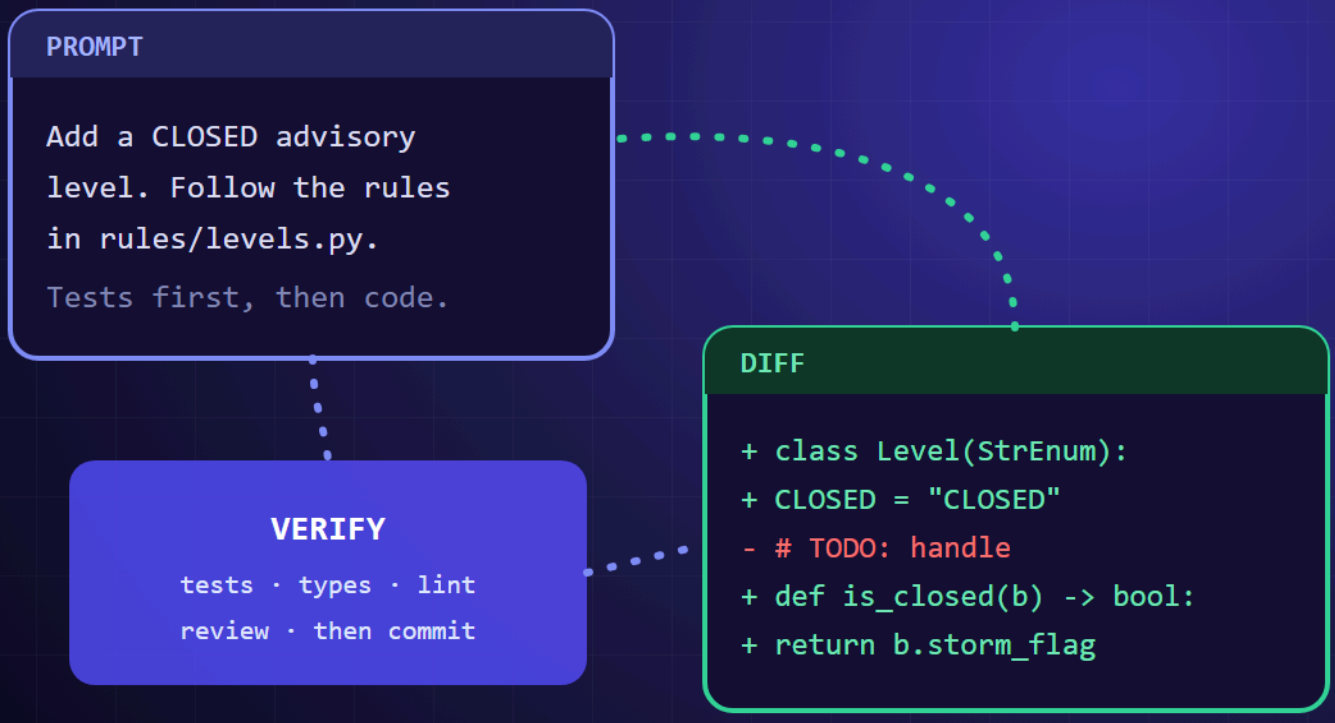




ASK · READ · TEST · REVIEW · OWN THE RESULT

A PRACTICAL GUIDE FOR SOFTWARE TEAMS

AI-Assisted Coding

Working effectively with AI coding assistants in real projects

28 chapters · 403 pages · one realistic codebase
from first prompt to production incident

PROMPTS · REVIEW · TESTING · SECURITY · AGENTS · TEAMS

THENEXBOOKS

AI-Assisted Coding

Working Effectively with AI Coding Assistants in Real Projects

A practical guide for professional software teams

First Edition

THENEXBOOKS

AI-Assisted Coding
Working Effectively with AI Coding Assistants in Real Projects

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

This book is deliberately written without naming specific assistant products, model versions, or vendors, because their capabilities, defaults, and terms change faster than any book can track. Where a concrete identifier would normally appear, you will find a placeholder such as `<assistant>` or `<model-id>`. Always consult the documentation and terms of the tools your team actually uses.

Chapter 24 discusses licensing, intellectual property, privacy, and compliance in general terms only. **Nothing in this book is legal advice.** Consult qualified counsel for your jurisdiction, your contracts, and your obligations.

Product and company names mentioned in this book are trademarks of their respective owners. Their use is for identification only and does not imply endorsement. This book is independent and is not affiliated with, sponsored by, or endorsed by any tool vendor, model provider, or platform.

Tidewell, the running example in this book, is fictional, as are the coastal authorities, data feeds, laboratories, and organizations mentioned in its examples. Any resemblance to real products or organizations is coincidental.

Preface

Something changed in the daily work of writing software, and it changed quickly. An assistant now sits inside the editor, in the terminal, and in the pull request. It completes the line you were about to type, drafts the function you were dreading, explains the module nobody on the team understands anymore, and — if you let it — runs commands and edits files on its own. For a great many developers this arrived as a set of impressive demos followed by a much harder question: *how do I actually work this way, on a real codebase, without shipping something I regret?*

That question is what this book answers. It is not a tour of features, and it is not an argument about whether any of this is good or bad. It is a practical manual for professional developers who already know how to build software and now want a reliable, honest way to do it with an assistant in the loop.

The premise running through every chapter is simple: **an assistant changes how you spend your attention, not whether you are responsible for the result.** Generated code is your code the moment you commit it. Everything in this book — how you frame requests, how you read a diff, how you test, review, secure, and measure — follows from taking that responsibility seriously.

Who This Book Is For

- **Working developers** who have an assistant available and want to go from ad-hoc prompting to a dependable workflow.
- **Senior engineers and tech leads** who must set conventions, review generated diffs, and keep quality steady across a team.
- **Engineering managers** deciding how assistance fits their process, and how to tell whether it is helping.
- **Developers joining an assisted team** who need to understand the norms, the risks, and where their own judgment is irreplaceable.

No machine-learning background is required. This is a book about engineering practice, not about how models are built.

What You Will Learn

- A working mental model of what assistants do, why they fail the way they do, and where their leverage is real.
- How to write requests that produce correct code, and how to specify work so the result can be verified.
- How to read a generated diff critically and find the bugs that look like good code.
- Complete, realistic feature work: an endpoint, a zero-downtime migration, tests with teeth, a legacy refactor, a production bug, and a dashboard screen.

- Review, testing, typing, security, and performance practices for a codebase where much of the diff was generated.
- Team conventions, instruction files, onboarding, the legal and privacy questions to ask early, and how to measure impact without fooling yourself.
- Agents and automation, huge codebases and long-lived migrations, and a full case study from planning to production incident to retrospective.

How This Book Is Organized

Part I — Foundations of Assisted Development (Chapters 1–5) builds the mental model, the workflow loop, the tooling and guardrails, the role of context, and introduces Tidewell. **Part II — Asking Well and Judging the Answer** (Chapters 6–10) covers request design, specifications, iteration, critical reading, and knowing when to take over. **Part III — Building Real Features** (Chapters 11–16) does six complete pieces of work in the codebase. **Part IV — Quality, Review, and Correctness** (Chapters 17–21) covers review, testing strategy, guardrails, security, and performance. **Part V — Teams, Process, and Responsibility** (Chapters 22–25) moves from personal practice to team assets, people, obligations, and measurement. **Part VI — Scaling Up and Shipping** (Chapters 26–28) covers agents, very large codebases, and a closing case study. Four appendices give you a prompt-pattern cheat sheet, reusable templates, a production readiness checklist, and a glossary.

Each chapter ends with a **Key Takeaways** box for quick review.

The Running Example: Tidewell

Throughout the book we work in **Tidewell**, a fictional coastal water-safety advisory service. It ingests tide predictions, weather observations, and water-quality lab samples, applies a set of rules, and publishes an advisory for each beach: safe, caution, no swimming, or closed. Supervisors can override an advisory; subscribers get notified; the public reads it on a beach page.

We chose it because it is small enough to hold in your head and messy enough to be honest. It is a modest monorepo — a Python API, a Python ingest worker, a rules engine, a TypeScript dashboard, SQL migrations, and a small Go operator CLI. It has genuine technical debt: a rule function that grew too long, thresholds hard-coded in three places, a slow integration test, one reliably flaky test, and a timezone assumption waiting to cause an incident. Every technique in this book gets applied to that code, not to a toy.

A Note on Tools and Versions

Assistant capabilities, model behavior, pricing, defaults, and terms change constantly. A book that named products and versions would be wrong within months and misleading immediately. So this book teaches mechanisms and practices that survive those changes, and uses placeholders such as

`<assistant>` where a specific name would otherwise go. Where behavior depends on your tool, we say so and point you at its documentation.

For the same reason, you will find no productivity statistics here. Claims about how much faster assisted teams are vary wildly by study, task, and measurement method. Chapter 25 shows you how to measure your own team instead of quoting someone else's numbers.

> **TIP**
> A recommended practice or a technique that makes your work easier.

> **NOTE**
> Background information, a clarification, or a useful aside.

> **WARNING**
> A common mistake or a behavior that can cause bugs, security holes, or outages.

Code appears in `monospace`. Prompts are shown in code blocks captioned `Prompt`, and generated output in blocks captioned `Assistant reply`, so you can always tell which is which. Long lines are wrapped to fit the page.

Contents

Preface

PART I — FOUNDATIONS OF ASSISTED DEVELOPMENT

1 What AI Coding Assistants Actually Do — and What They Do Not

2 The Shape of a Good Assisted Workflow

3 Setting Up: Editors, CLI Agents, Review Bots, and Guardrails

4 Context Is Everything: How an Assistant Sees Your Repository

5 Meet Tidewell: The Codebase We Will Work In

PART II — ASKING WELL AND JUDGING THE ANSWER

6 Writing Requests That Produce Correct Code

7 Specifications, Acceptance Criteria, and Definition of Done

8 Iterating in Small Steps with Fast Feedback

9 Reading Generated Code Critically

10 Knowing When to Stop Prompting and Write It Yourself

PART III — BUILDING REAL FEATURES

11 From Ticket to Working Endpoint

12 Data Modeling and Database Migrations

13 Generating Tests That Actually Catch Bugs

14 Refactoring Legacy Code Without Breaking It

15 Debugging: From Stack Trace to Root Cause

16 Frontend Work: Components, State, and Accessibility

PART IV — QUALITY, REVIEW, AND CORRECTNESS

17 Code Review When Half the Diff Was Generated

18 Testing Strategy in an Assisted Codebase

19 Types, Linters, and Automated Guardrails

20 Security Review of Generated Code

21 Performance Work: Measure First, Then Ask

PART V — TEAMS, PROCESS, AND RESPONSIBILITY

22 Team Conventions: Instruction Files, Rules, and Shared Prompts

23 Onboarding, Pairing, and Keeping Knowledge in the Team

24 Licensing, Intellectual Property, Privacy, and Compliance

25 Measuring Impact Without Fooling Yourself

PART VI — SCALING UP AND SHIPPING

26 Agents: Multi-Step Tasks, Tools, and Automation in CI

27 Large Codebases, Monorepos, and Long-Lived Migrations

28 Case Study: Shipping Tidewell v2 with an Assisted Team

APPENDICES

A Prompt Patterns Cheat Sheet

B Templates and Reference

C Production Readiness Checklist

D Glossary

PART I

Foundations of Assisted Development

What a coding assistant actually is, the loop that makes it useful instead of dangerous, the tools and guardrails worth setting up on day one, how context reaches the model, and a tour of Tidewell — the codebase we will work in for the rest of the book.

1 · What AI Coding Assistants Actually Do
2 · The Shape of a Good Assisted Workflow
3 · Setting Up: Editors, Agents, and Guardrails
4 · Context Is Everything
5 · Meet Tidewell

CHAPTER 1 · PART I — FOUNDATIONS OF ASSISTED DEVELOPMENT

What AI Coding Assistants Actually Do — and What They Do Not

An AI coding assistant is neither a compiler nor a colleague. It is a text predictor working from whatever fragment of your project happens to be in front of it. This chapter builds the mental model that makes every later technique in the book obvious rather than mysterious, and introduces Tidewell, the codebase we will work in.

Start With the Right Mental Model

Most of the frustration developers feel with AI coding assistants comes from an unstated assumption: that the thing on the other side of the chat box understands the program. It does not, at least not in the sense you mean when you say a teammate understands the program. What it does is far stranger and, once you accept it, far more predictable.

A coding assistant is a statistical model of text. It has been trained on an enormous quantity of writing — source code, documentation, issue threads, tutorials, commit messages, books — and from that training it has learned which sequences of tokens tend to follow which other sequences. When you ask it for a function, it is not consulting a specification and deriving an implementation. It is producing the most plausible continuation of the text you gave it, one chunk at a time, where "plausible" means "resembles the code that followed similar text in its training data".

This is not a criticism. Plausibility, learned from a large enough corpus of real code, turns out to be shockingly useful. Most of the code any of us write is not novel; it is a variation on a pattern that thousands of other developers have written before. A model that is extremely good at pattern completion will therefore be extremely good at large parts of your job. But plausibility is not correctness, and the gap between them is where every problem in this book lives.

It writes forward, and it cannot take anything back

One more mechanical detail earns its place in the model, because it explains a behavior that otherwise looks like stubbornness. Output is produced incrementally, piece by piece, and each piece is chosen partly at random from the plausible candidates. Once a piece is committed it becomes part of the text the model is continuing from. There is no revision pass in which the model reconsiders its opening line.

The practical effect is that early choices cascade. If the first line of a generated function commits to iterating over a list, the rest of the function will be built around iterating over a list, even when a dictionary lookup was the better design. If the opening sentence of an explanation asserts something false, later sentences will elaborate on the falsehood rather than notice it, because elaborating is what

plausible continuations of an assertion look like. This is why asking a model to "check your previous answer" sometimes produces a genuine correction and sometimes produces a confident defense: the earlier answer is now context, and context is persuasive.

It is also why the single most effective intervention, when an answer goes off the rails, is not to argue. It is to discard the attempt, improve the framing, and start again. A fresh attempt with better context beats three rounds of correction more often than feels fair.

The context window is the whole world

The second half of the mental model concerns what the model can see. Every request you make is turned into a single body of text — your instructions, some portion of your code, perhaps a rules file, perhaps the earlier turns of the conversation — and that body of text has a size limit. This limit is usually called the **context window**. Nothing outside it exists as far as this particular request is concerned.

Think of it as a whiteboard of fixed size. Before the assistant answers, your tool scribbles onto that whiteboard everything it guesses will be relevant: the file you have open, a few snippets it retrieved by searching, the project's instruction file, the last several messages. Then the model reads the whiteboard and writes a continuation. It has no ability to walk down the corridor and look at a file that did not make it onto the board. If your validation rules live in a module nobody put there, the assistant will invent validation rules, and they will look exactly as confident as the correct ones would have.

Agentic tools blur this picture slightly, because an agent can run a search, read a file, and put the result onto the whiteboard itself before continuing. That is an enormous improvement, and Chapter 26 goes into it properly. But it does not change the fundamental constraint. It only means the assistant is now allowed to fetch things — and it still has to guess what to fetch, and the whiteboard is still finite. Chapter 4 is entirely about spending that budget well.

No memory, unless something gives it one

A new conversation starts from nothing. The assistant does not remember yesterday's debugging session, the architectural decision you explained last week, or the fact that your team banned a particular library. Any apparent memory comes from machinery around the model: a conversation transcript being replayed, an instruction file being injected, a summary being carried forward. When people say "the assistant forgot what we agreed", what actually happened is that the agreement was never written anywhere the tool could find it. Chapter 22 turns this into a team practice.

> **NOTE**
>
> Throughout this book we write about "your assistant", "the model", "a CLI agent", and "a review bot" rather than naming products. Tools change quickly, feature sets differ, and anything we claimed about a specific release would be stale before you read it. Where a concrete identifier is unavoidable in a configuration file, we use a placeholder such as `<assistant>` or `<model-id>`. Substitute whatever you actually use, and check its documentation for the details.

Four Consequences You Will Meet This Week

From "plausible continuation over a limited window" everything else follows. Four consequences matter enough to name.

1. Fluent and wrong is the default failure mode

Traditional tools fail loudly. A compiler refuses. A linter complains. A test goes red. An assistant fails *quietly and beautifully*. The wrong answer has the same crisp docstrings, the same tidy type hints, and the same confident tone as the right one. There is no tremor in the output to warn you.

Here is a Tidewell example. Suppose the advisory rules engine must return `NO_SWIM` when a water sample exceeds the bacterial threshold, but only if the sample was collected within the last 48 hours; older samples are treated as unknown rather than as evidence of safety. Ask for that without supplying the staleness rule and you will get something like this.

Assistant reply (excerpt)

```
def evaluate_water_quality(sample: WaterSample | None) -> Advisory | None:
    """Return a NO_SWIM advisory when bacteria exceed the safe limit."""
    if sample is None:
        return None
    if sample.bacteria_cfu_per_100ml > BACTERIA_LIMIT:
        return Advisory(level="NO_SWIM", reasons=["bacteria_exceeded"])
    return None
```

Read that as a reviewer for a moment. It is clean, typed, readable, and has a docstring that matches its behavior. It would sail through a casual review. It is also wrong in a way that matters: a sample from three weeks ago will be treated as current, and a stale-but-clean sample will silently produce no advisory at all, which downstream code interprets as "nothing to worry about". On a real service, that is a beach that should have been flagged and was not.

The model did not lie. It had no access to the staleness rule, so it completed the pattern it did have — threshold comparisons look like this in most code it has ever seen. The failure is a context failure, dressed as a logic failure.

2. Invented APIs

Because the model predicts plausible text, it will happily predict plausible *names*. If your project has `services/rules/thresholds.py` with a function `bacteria_limit_for(beach)`, and that file is not in context, an assistant asked to use "the threshold helper" may produce `get_bacteria_threshold(beach.region)` — a function that does not exist, imported from a module that does not exist, with an argument order nobody chose. The same happens with third-party libraries: a method that *ought* to exist by symmetry with its siblings often does not.

This class of error is the cheapest to catch and the most embarrassing to ship. An import error surfaces the moment you run anything. A missing attribute on an object surfaces the first time that branch executes — which, if the branch is an error path, may be in production at 3 a.m. Type checking and a test that actually exercises the new code close this gap almost entirely, which is why Chapter 3 spends so much time on making both of those instant to run.

3. The training cutoff and deprecated advice

A model's training data stops at some point. Everything after that point — new library releases, renamed APIs, changed defaults, newly discovered pitfalls — is invisible unless it reaches the context window some other way. In practice this shows up as suggestions that were idiomatic a version or two ago.

Tidewell's API service uses Pydantic v2. An assistant whose training leaned on older material may reach for v1 idioms without hesitation.

Assistant reply (excerpt) — outdated style

```
class AdvisoryOut(BaseModel):
    beach_slug: str
    level: str

    @validator("level")           # v1 decorator
    def check_level(cls, v):
        assert v in LEVELS
        return v

    class Config:                 # v1 configuration style
        orm_mode = True

payload = advisory_out.dict()     # v1 serialization method
```

The current equivalent

```
class AdvisoryOut(BaseModel):
    model_config = ConfigDict(from_attributes=True)

    beach_slug: str
    level: AdvisoryLevel

    @field_validator("level")
    @classmethod
    def check_level(cls, v: AdvisoryLevel) -> AdvisoryLevel:
        if v not in LEVELS:
            raise ValueError(f"unknown advisory level: {v}")
        return v

payload = advisory_out.model_dump()
```

Note that the outdated version is not gibberish. It is correct code for a library version your project does not use. That is the characteristic shape of cutoff-related errors: internally consistent, externally stale. The defense is not to argue with the model about versions — it has no way to check — but to make your project's actual conventions visible. A pinned lockfile, a strict type checker, and two or three exemplar files pointed at explicitly will fix this far more reliably than any amount of prompting.

> **WARNING**
>
> Deprecated suggestions are especially dangerous in security-adjacent code, where yesterday's recommended practice is sometimes today's known weakness. Password hashing parameters, token validation, TLS options, and cryptographic defaults must be checked against current documentation rather than accepted because they look conventional. Chapter 20 treats this in detail.

4. Non-determinism

Run the same prompt twice and you may get different code. Sometimes the difference is cosmetic — a renamed variable, a comprehension instead of a loop. Sometimes it is structural, and sometimes one version is correct while the other is not. Two answers to "parse the tide feed timestamp" might look like this.

Two runs, same prompt

```
# run A
observed_at = datetime.fromisoformat(row["ts"]).astimezone(UTC)

# run B
observed_at = datetime.strptime(row["ts"], "%Y-%m-%dT%H:%M:%SZ")
observed_at = observed_at.replace(tzinfo=UTC)
```

Run A handles offsets such as `+01:00` and fails on a feed that omits the offset entirely. Run B hard-codes the `Z` suffix and throws on anything else. Neither is unreasonable; which one is correct depends on what `NATIONAL-TIDE-GRID` actually emits, and on what you want to happen when it emits something unexpected. The model cannot know that. You can, and a test with three sample rows in it encodes the answer permanently.

Non-determinism has a practical consequence that is easy to miss: **a prompt is not an artifact you can rely on**. You cannot check a prompt into version control and expect it to reproduce a build. What you check in is code, tests, types, and configuration. The prompt was scaffolding. This is the single biggest mental adjustment for developers who come to assisted coding hoping for a declarative workflow: the determinism has to live downstream of the model, in your verification, not upstream in your wording.

What an Assistant Is Not

It is worth being blunt about four analogies that mislead.

- **Not a compiler.** It does not parse, type-check, or execute anything. Output that looks like it must have been validated has not been. Even an agent that runs your tests only knows what the test output said; it did not reason its way to correctness first.
- **Not a search engine.** It does not look things up unless it has a retrieval or web tool and chooses to use it. A confident API reference from a plain chat turn is a reconstruction from memory, not a lookup, and reconstructions blur.
- **Not a colleague with memory.** There is no shared history, no accumulated understanding of why the ingest worker retries three times, no institutional knowledge. Everything must be re-supplied or written down somewhere the tool reads.
- **Not a source of truth about your repository.** When it says "the existing handler already validates the slug", that is a prediction about what such a handler probably does, unless the handler is genuinely in context. Verify before believing.

The positive framing is more useful than the negative one: an assistant is an extremely fast, extremely well-read, slightly careless collaborator with no memory and no access to anything you do not hand it. Hire that person and you would still review their work. You would also quickly learn to brief them well, which is most of what this book teaches.

The Shapes Assistants Come In

"AI coding assistant" covers at least four quite different tools. They share a model and almost nothing else. Choosing the right shape for a task is a skill in itself, and using the wrong one accounts for a surprising share of bad experiences.

```
scope of a single action        supervision required
-------------------------------------------------------------
inline completion    a line or two        glance
editor chat          a function/file      read the diff
CLI agent            many files, cmds     approve + review
CI review bot        a whole pull request  triage comments
```

Inline completion

A suggestion appears ahead of your cursor as you type; you accept it with a keystroke. The context is mostly the current file plus whatever nearby files the tool samples. This shape is best when you already know what you want to write and the typing is the slow part: filling in a data class, writing the fifteenth similar test case, completing a familiar call.

Its weakness is that acceptance is nearly frictionless. A keystroke is cheaper than reading, so inline completion quietly trains you to stop reading. It is also poor at anything needing a decision, because it optimizes for the most likely next tokens, and "most likely" and "correct for this beach's edge case" are

different targets. Watch for the specific trap of completion-driven design, where you keep the structure the tool suggested because rejecting it costs more keystrokes than accepting it.

Chat in the editor

A side panel where you describe what you want, referencing files and selections. The context is larger and, crucially, you control much of it. This is the workhorse shape for most feature work: you can state intent, name the two files that define the pattern to copy, paste a failing test, and get a candidate you then review as a diff.

Its weakness is conversational drift. After a dozen turns the transcript contains half-abandoned ideas, superseded code, and instructions you no longer want followed, and the model weighs all of it. Starting a fresh conversation is often faster than correcting a confused one — a habit worth building early.

A CLI agent

A terminal program that can read and write files, run commands, and loop: plan, act, observe the output, act again. This is the shape with the most leverage and the most risk. Given "make `pytest services/rules` pass after the threshold refactor", an agent can find the failures, edit several files, re-run the suite, and iterate — work that is tedious and mechanical for a human.

The risks are proportionate. An agent runs commands, and commands have consequences: deleting files, rewriting history, dropping a database, calling an external service. It can also flail productively-looking for a long time, producing a large diff full of unrelated changes. Sandboxing, explicit approval for irreversible actions, and a hard habit of reviewing the diff before committing are not optional. Chapter 3 sets up the guardrails and Chapter 26 covers agents properly.

A review bot in CI

A bot that reads a pull request diff and leaves comments. It sees the change and usually some surrounding code, but rarely the whole system or the ticket's intent. It is good at the tireless-reviewer layer: a missing `await`, an unhandled branch, a test that asserts nothing, an obvious injection risk, an inconsistency with a neighboring file.

It is bad at judgment. It cannot tell you that this endpoint should not exist, that the approach conflicts with a decision the team made in March, or that the feature is fine but the migration will lock a hot table. It also produces false positives, and a bot that comments on everything trains reviewers to ignore it. Treat its output as a checklist to triage, never as an approval. Chapter 17 covers this.

Shape	Strong at	Weak at
Inline completion	Typing you already planned; repetitive blocks	Decisions; edge cases; it is too easy to accept
Editor chat	Functions, small modules, explanations, tests	Long threads drift; limited to what you show it

CLI agent	Multi-file mechanical work; test-driven loops	Unsupervised commands; sprawling diffs
CI review bot	Consistent low-level checks on every change	Intent, design, cross-system risk; noise

> **TIP**
>
> Match the shape to the blast radius. If the change is one line, use completion. If it is one function, use chat. If it is twelve files of mechanical edits with a test suite to prove them, use an agent. If it is a judgment call about architecture, use your own head and ask the assistant to argue against you afterwards.

Where the Real Leverage Is

Assistants are not uniformly useful. The returns are concentrated in tasks where the answer is *conventional but tedious*, or where your bottleneck is unfamiliarity rather than thinking. Five categories stand out.

- **Boilerplate and mechanical translation.** Turning a table definition into a Pydantic model and a TypeScript interface, writing the serializer, wiring the router, adding the migration skeleton. There is a single right shape, it is well represented in training data, and a type checker verifies it instantly.
- **Unfamiliar APIs and languages.** Tidewell's `tools/tidectl/` is Go, and most of the team writes Python. Asking for idiomatic error handling or a `cobra`-style subcommand skeleton saves an hour of documentation reading — provided you verify against the real docs, since this is exactly where invented APIs appear.
- **Test scaffolding.** Parametrized cases, fixtures, fake feed payloads, builders for a `Beach` with sensible defaults. Enumerating the boring half of a test matrix is something a model does quickly and you do grudgingly. The cases that actually matter still need your judgment; Chapter 13 draws that line carefully.
- **Refactors with a safety net.** Renaming a concept across thirty files, extracting an interface, threading a new parameter. The word "net" is load-bearing: the value comes from a green test suite you trusted *before* the refactor started.
- **Explaining unfamiliar code.** Point at the 600-line ingest scheduler nobody wants to touch and ask what it does, what the failure modes are, what the retry semantics are. Explanations are cheap to verify — you can read the code with the summary as a map, and a wrong summary tends to be obviously wrong once you look.

Notice what these have in common: either the correct answer is highly constrained, or a mistake is cheap and immediately visible. That is the real selection criterion, and it is worth applying deliberately rather than by feel.

Where It Is Weakest

The mirror image is equally concentrated.

- **Novel algorithms.** If nothing resembling your problem exists in the training corpus, pattern completion produces something that looks like an algorithm and is not one. Tidewell's advisory precedence logic — how a supervisor override interacts with an automatic escalation and an expiring sample — is genuinely ours, and the assistant will confidently guess wrong at it.
- **Cross-cutting architecture.** Whether the rules engine should own scheduling, whether advisories should be event-sourced, how to split the monorepo. These depend on constraints that mostly live in people's heads: team size, on-call tolerance, the migration you already half-finished. An assistant is a useful sparring partner here and a poor decision-maker.
- **Anything requiring facts not in the repository.** The regulator's actual reporting deadline. Why the previous team pinned that dependency. What `HARBORLAB` returns when a sample is rejected. A model asked a question whose answer is not available will produce an answer anyway. This is the single most common source of confidently wrong output.
- **Security-critical logic.** Authorization checks, token handling, tenant isolation, anything touching secrets. The failure mode is code that passes tests and is still exploitable. Generated code here needs a threat-model review, not just a test run.
- **Judging its own work.** An assistant that wrote the implementation and then wrote the tests has tested its own assumptions. Both can be wrong together and agree perfectly, and a green suite will make you feel safe. Chapter 2 names this as an anti-pattern for good reason.

The Core Claim of This Book

Here is the thesis everything else rests on: **an assistant changes how you spend your attention, not whether you are responsible for the result.**

Before, your attention went into production: recalling syntax, typing, looking up signatures, writing the fifth similar test. Now much of that is cheap, and attention moves to *specification* (saying precisely what you want, with edge cases), *context* (making sure the relevant code and constraints are visible), and above all *verification* (proving the candidate is right before it becomes yours).

The total effort does not necessarily drop. What changes is its shape, and the shape is less comfortable, because reviewing is harder to stay alert through than writing. Writing has natural friction that forces engagement; reading fluent code that is probably fine does not. Much of the discipline in this book exists to replace that lost friction with deliberate checks.

Responsibility does not move at all. When a bad advisory reaches a lifeguard team, "the assistant wrote it" is not an explanation, a defense, or an incident report. Your name is on the commit. That is not a reason for anxiety; it is a reason for a workflow, which is exactly where Chapter 2 goes.

Meet Tidewell

Every technique in this book is demonstrated on one codebase, so that examples accumulate instead of resetting each chapter. **Tidewell** is a coastal water-safety advisory service. Coastal authorities and lifeguard teams use it to answer one question for each beach they manage: is it safe to swim there today?

It ingests tide and weather feeds on a schedule, receives laboratory water-sample results, runs a rules engine that combines them into an advisory level — `SAFE`, `CAUTION`, `NO_SWIM`, or `CLOSED` — publishes that through a public API, lets supervisors override it through a dashboard, and notifies subscribers when it changes. The repository is a small monorepo.

Repository layout

```
tidewell/
  services/api/        Python 3.12 + FastAPI: public and internal API
  services/ingest/     Python worker: pulls external feeds on a schedule
  services/rules/      the advisory rules engine (pure, heavily tested)
  web/dashboard/       TypeScript + React admin console for supervisors
  db/migrations/       SQL migrations (PostgreSQL)
  tools/tidectl/       a small Go CLI used by operators
```

The domain is small enough to hold in your head and awkward enough to be realistic. A `Beach` has a slug such as `cobble-bay`, a region, coordinates, a patrol flag, and a season window. Against each beach we record `TideReading` and `WeatherReading` rows from the feeds, and `WaterSample` rows from the laboratory with a bacterial count and a status. The rules engine turns those into an `Advisory` carrying a level, a list of machine-readable reason codes, an issue time, an expiry, and whether it was issued by the system or by a supervisor. `Subscriber` records say who is told about changes and over which channel, and an audit trail records every advisory decision because somebody will eventually ask why a beach was closed on a Tuesday.

Three external feeds — all fictional, so nothing in this book should be read as a claim about a real organization — supply the raw data: `NATIONAL-TIDE-GRID` for tides, `COASTWATCH-METEO` for weather, and `HARBORLAB` for water samples. Each one is unreliable in its own characteristic way, which is precisely why they are useful teaching material.

Tidewell was chosen because it exercises everything an assisted workflow touches: scheduled ingestion of feeds you do not control, a rules engine with genuinely tricky edge cases, a public contract you cannot break, database migrations, a frontend, flaky tests, on-call incidents, and a pile of legacy code nobody wants to open. Chapter 5 gives it a proper tour, including the parts that are ugly on purpose. For now, you only need to know that when a chapter says "ask the assistant to add a reason code to the advisory response", there is a real shape of code behind that request.

You do not need to build Tidewell to use this book. Read the examples as a stand-in for the system you actually work on, and the mapping will be obvious — everyone has an ingest job they distrust and a module they inherited.

Key Takeaways

- An assistant predicts plausible text over a limited context window; it does not compile, execute, look up, or remember anything by itself.
- Its default failure mode is fluent and wrong — clean, typed, documented code that is subtly incorrect because a rule was never in context.
- Expect invented functions and modules, suggestions aimed at older library versions, and different code from the same prompt on a second run.
- A prompt is not a reproducible artifact. Determinism has to come from tests, types, and review, not from wording.
- Four shapes — inline completion, editor chat, CLI agent, CI review bot — have genuinely different strengths; match the shape to the blast radius.
- Leverage is highest where the answer is conventional and mistakes are cheap: boilerplate, unfamiliar APIs, test scaffolding, netted refactors, explanations.
- Leverage is lowest for novel algorithms, cross-cutting architecture, facts not present in the repository, and security-critical logic.
- The assistant changes where your attention goes; it does not move responsibility for the result away from you.